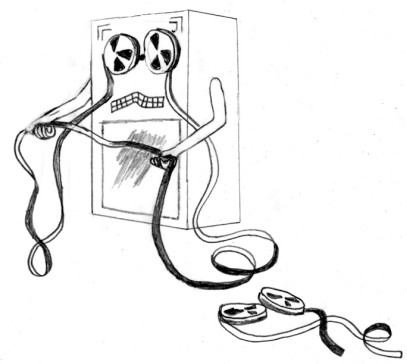
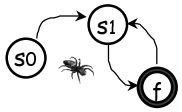
from code to models



gerard j. holzmann
computing principles research
bell laboratories, usa

code, abstraction, and analysis



- the intent of a formal model is to enable analysis by preserving selected real-world artifacts, and suppressing others
  - a model is a deliberate simplification of a problem
  - it captures the minimal assumptions necessary to establish facts
- formal models must have refutation power
  - "the purpose of analysis is not to compel belief, but rather to suggest doubt"
    - Imre Lakatos (1922-1974)
  - "seek simplicity, and distrust it."
    - Alfred North Whitehead (1861-1947)

2

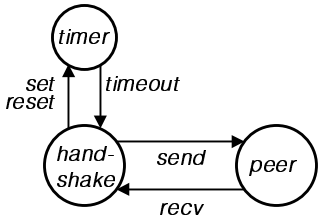
wish-list



- to be effective in software development:
  - a verification model must enable a form of analysis that cannot be obtained more easily by other means
  - the construction of the model must be (perceived as) a minor task, compared with the building of the actual system
- model construction relies on abstraction
  - the abstraction can be captured in rules
  - these rules must be explicitly documented and tracked
  - they must be able to evolve with the code
  - model construction is ideally automated

3

example




```
extern const int p0;
enum mtype { Msg, Ack, TimeOut, Other, Set, Reset };

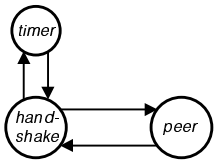
void
handshake(void)
{       int resp;

        send(p0, Msg);
        set_timer(16000); /* msec */

        resp = wait_recv();
        switch (resp) {
        case Ack:
                reset_timer();
                /* . . . */
                break;
        case TimeOut:
                /* . . . */
                break;
        default:
                reset_timer();
                error("bad response");
                break;
        }
}
```

4

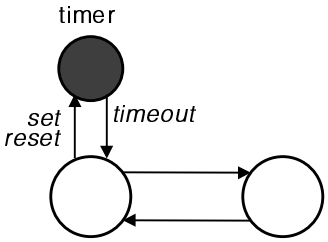
verification context



- software verification is always done in a context
  - the context is often left implicit, leading to unstated assumptions
- the context captures minimal assumptions about
  - device behavior
  - user interactions, user input
  - components that are not the focus of the verification
  - correctness properties (requirements for correctness)

5

context 1: timer behavior



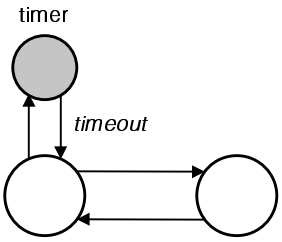

```
/*
 * attempt to duplicate the behavior
 * of a count-down timer
 */
active proctype timer_p()
{       chan who;
        short cnt;

        do
        :: timer?Set(who,cnt) ->
                do
                :: timer?Reset(who,cnt) ->
                      break
                :: empty(timer) ->
                        if
                        :: cnt > 0 -> cnt--
                        :: else ->
                                who!TimeOut;
                                break
                        fi
        od      od
}

2^16 states (16,001 reachable)
```

6

abstraction of timer




```
/*
 * the relative speeds of asynchronous
 * processes is unknown and unknowable.
 * the external behavior of timer_p is
 * indistinguishable from this:
 */

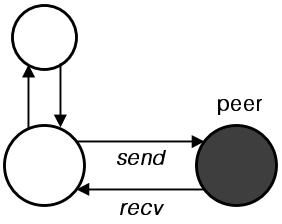
active proctype timer_p()
{       chan who = 0;

        do
        :: timer?Set(who,_)
        :: timer?Reset(who,_) -> who = 0
        :: who != 0 -> who!Timeout
        od
}
```

2 states

7

context 2: remote peer



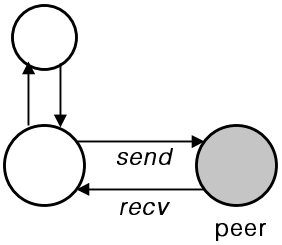

```
/*
 * handshake expects to receive
 * an integer value - send one
 */
active proctype peer()
{       int n;

        do
        :: p0?Msg ->
                if
                :: q0!Ack
                :: n = 0;
                        do
                        :: n++
                        :: break
                        od;
                        q0!n
                fi
        od
}
```

> 2^32 states

8

abstraction of peer




```
/*
 * handshake process can only
 * distinguish between two
 * types of responses
 * define an abstraction for these
 * response types (int -> enum)
 */

active proctype peer()
{
        do
        :: p0?Msg ->
                if
                :: q0!Ack
                :: q0!Other
                fi
        od
}
```

2 states

9

model construction:
first attempt - a hand-built model

```
chan p0 = [0] of { mtype };
chan q0 = [0] of { short };
chan timer = [0] of { mtype, chan, short };

mtype = { Msg, Ack, Other, Set, Reset, TimeOut };

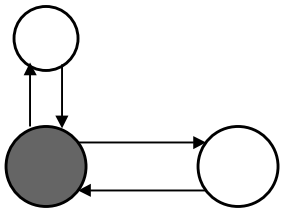
active proctype handshake()
{       short resp;

        p0!Msg;
        timer!Set(p0,16000);
        q0?resp;
        if
        :: resp == Ack -> timer!Reset(0,0)
        :: resp == TimeOut ->
        :: else -> timer!Reset(0); assert(false)
        fi
}
```

10

model extraction

mechanically generated model (before abstraction)



```
hidden int TimeOut = 1;
hidden int Ack = 2;
hidden int Msg = 3;

int p0;

active proctype handshake()
{   int resp;

    c_code { send(now.p0,Msg); };
    c_code { set_timer(16000); };
    c_code { Phandshake->resp = wait_recv(); };

    do
    :: c_expr { Ack == Phandshake->resp };
         c_code { reset_timer(); };
         break; goto C_0
    :: c_expr { TimeOut == Phandshake->resp };
C_0:     break; goto C_1
    :: else ->
C_1:     c_code { reset_timer(); };
         c_code { error("bad response"); };
         break; goto C_2
     od;
C_2: skip;
}
```

11

matching the extracted model into the target verification context

```
set_timer(16000)
reset_timer()
send(p0,Msg)
resp = wait_recv()
error("bad response")
resp == Ack
resp == Timeout
```

context dependent primitives in code fragment



12

matching the extracted model into the verification context

```
set_timer(16000)        ⟶   timer!Set(q0,1600)
reset_timer()           ⟶   timer!Reset(q0,0)
send(p0,Msg)            ⟶   p0!Msg
resp = wait_recv()      ⟶   q0?resp
error(…                 ⟶   assert(false)
```

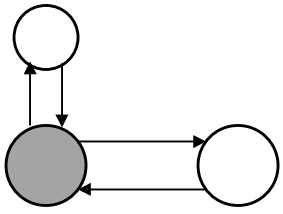
pattern rule

- *syntax conversions*
- *abstraction rules*
- *explicit assumptions*
- *tracking changes*

13

model extraction

with abstraction



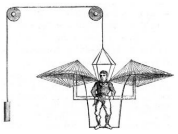
```
active proctype handshake()
{   int resp;

    p0!Msg;
    timer!Set(q0,16000);
    q0?resp;

    do
    :: c_expr { Ack == Phandshake->resp };
         timer!Reset(q0,0);
         break; goto C_0
    :: c_expr { TimeOut == Phandshake->resp };
C_0:     break; goto C_1
    :: else ->
C_1:     timer!Reset(q0,0);
         assert(false);
         break; goto C_2
     od;
C_2: skip;
}
```

14

correctness properties



- temporal logics (CTL, LTL, …)
  - powerful, expressive
  - but sometimes non-intuitive, even for experts
- alternative: exploit formalisms already used by programmers
  - assertion libraries
  - visual formalisms

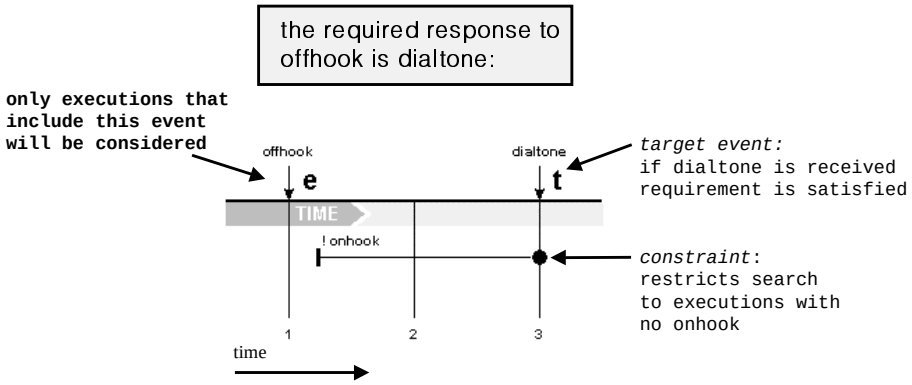
15

FeaVer assertion library

- L: assert(p) *invariance*
  - when L is reached expression p is always true
    - $\Box$ (When_At_L $\rightarrow$ p)
- L: assert_r(p, q) *response*
  - when L is reached expression p is always true and
  - p remains true at least until q becomes true
    - $\Box$ (When_At_L $\rightarrow$ (p U q))
- L: assert_p(p) *precedence*
  - when L is reached expression p always becomes true
  - within a finite number of steps
    - $\Box$ (When_At_L $\rightarrow \Diamond$ p)

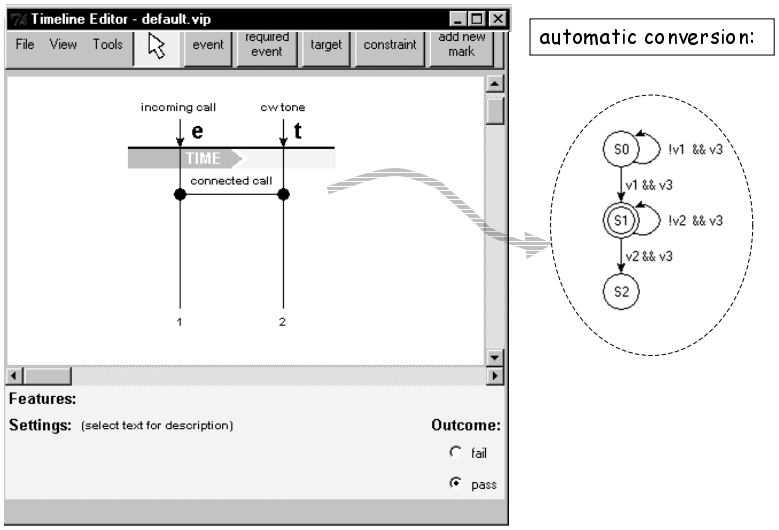
16

FeaVer visual timeline editor



17

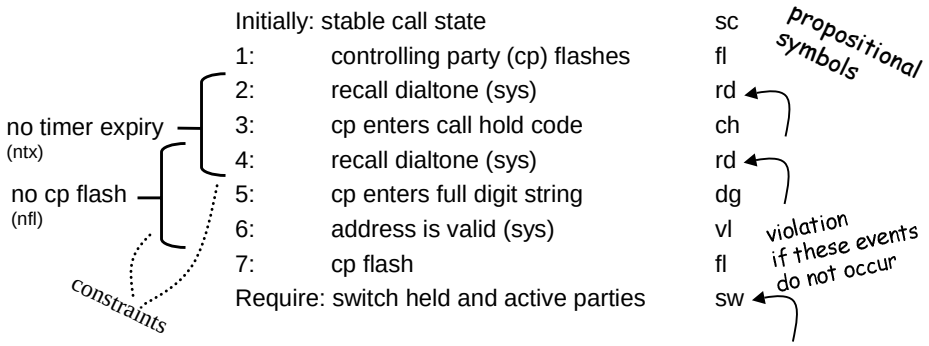
semantics



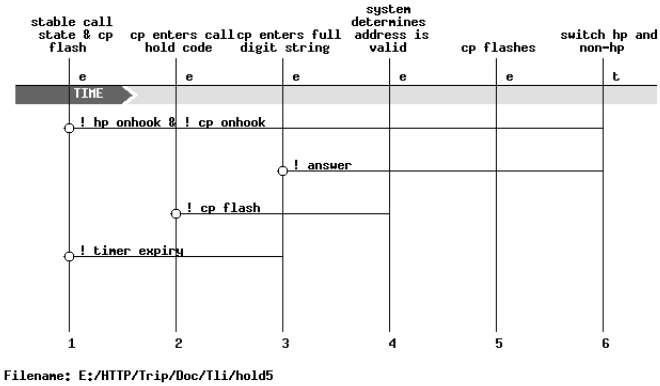
18

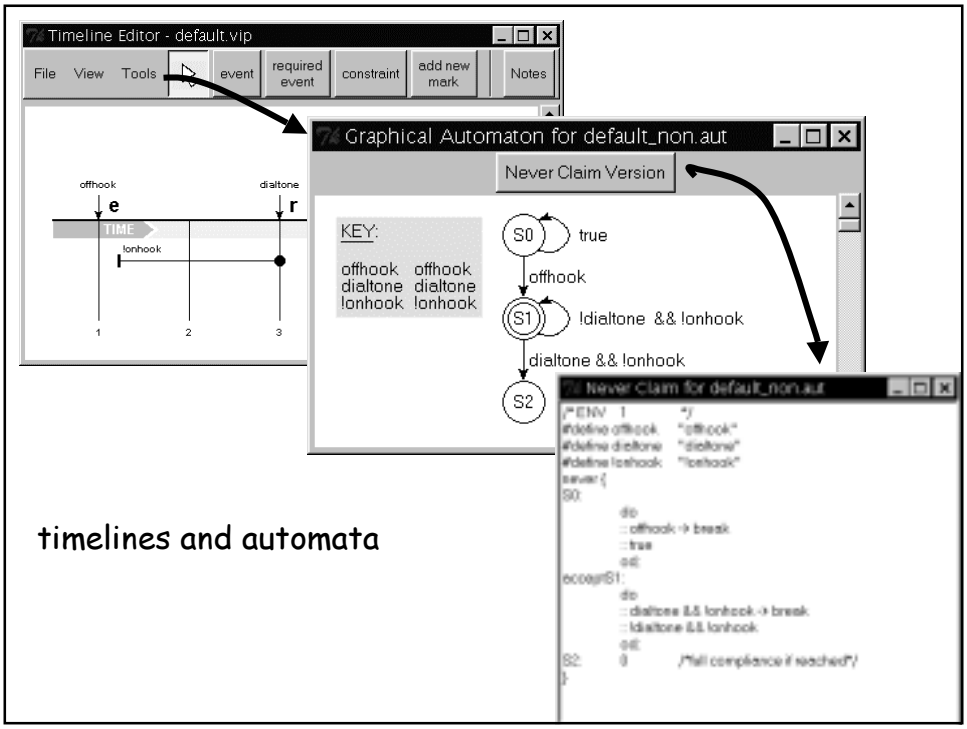
event and constraints

call hold: *a cp flash must switch connected and held party*



call hold requirement





timelines and automata

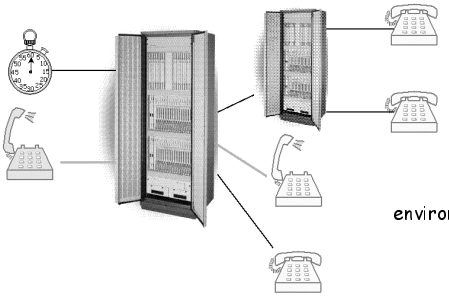
timelines and ltl

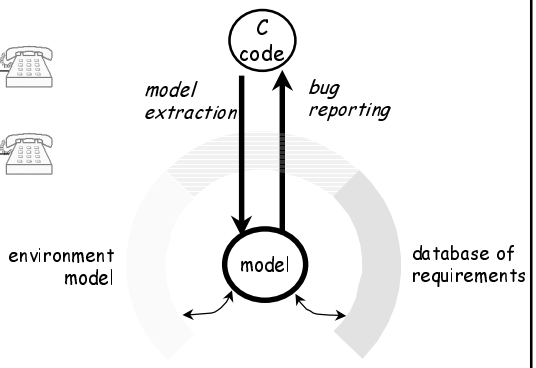


- a timeline with *k+1* events requires an LTL formula with *until-depth* ≥ *k*.
  - formula will have *k* nested subformulae
- stutter-invariance is not guaranteed
  - stutter-invariance can be exploited by Spin in its partial order reduction algorithm.
  - for timelines, we use algorithmic checks on the generated automata to determine if a property is stutter-invariant.

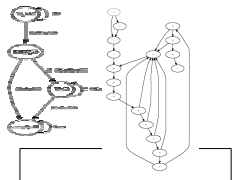
22

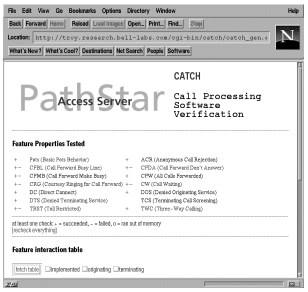
model extraction of real code

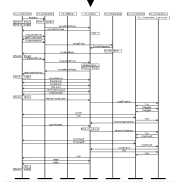




the feaver system

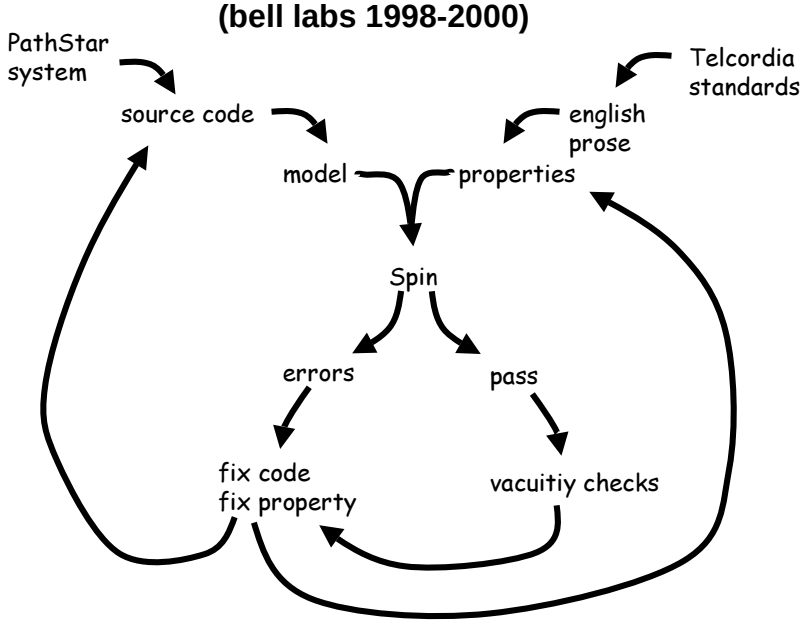








the pathstar project
(bell labs 1998-2000)



25

summary



- abstraction is key to software verification
  - computational complexity is a given
  - abstraction enables analysis
  - no different from any other engineering discipline
- there are great opportunities to merge a number of relatively mature fields in mechanically deriving formal models from source code
  - static analysis / program analysis / slicing
  - logic model checking
  - theorem proving

26